Continuous Assessment of a Unix Configuration: Integrating Intrusion Detection and Configuration Analysis

Abdelaziz Mounji and Baudouin Le Charlier

Institut d'Informatique
University of Namur
rue Grandgagnage 21
B-5000 Namur Belgium
Email: {ble, amo}@info.fundp.ac.be

Abstract

Computer security is a topic of growing concern because, on the one hand, the power of computers continues to increase at exponential speed and all computers are virtually connected to each other and because, on the other hand, the lack of reliability of software systems may cause dramatic and unrecoverable damage to computer systems and hence to the newly emerging computerized society. Among the possible approaches to improve the current situation, expert systems have been advocated to be an important one. Typical tasks that such expert systems attempt to achieve include finding system vulnerabilities and detecting malicious behaviours of users.

In this paper, we extend our intrusion detection system ASAX with a deductive subsystem that allows us to assess the security level of a software configuration on a real time basis. By coupling the two subsystems — intrusion detection and configuration analysis — we moreover achieve a better tuning of the intrusion detection since the system has only to enable intrusion detection rules that are specifically required by the current state of the configuration. We also report some preliminary performance measurements, which suggest that our approach can be practical in real life contexts.

1. Introduction

Securing computers and computer networks against unauthorized access and misuse is a growing concern among the computer community as evidenced by the rising number of press reports about security incidents. Besides enforcing user identification, user authentication and data integrity validation, *Access control* provides mechanisms for preventing break-ins and for preventing authorized users from accessing resources they are not allowed to use. However, the adequate configuration of an access control system may be challenging for unexperienced administrators due to the complexity of the interactions between configuration decisions. To overcome this difficulty, several *configuration analysis systems* have been developed allowing to check for common vulnerabilities such as improperly exported file systems and bad permission modes for security sensitive files and directories. Based on the fact that access control mechanisms may be abused by intruders due to flaws in the access control software itself or to configuration mistakes, *intrusion detection expert systems* (IDES) have been advocated as a last line of defense against security violations. The IDES approach consists of observing system and user activities in order to detect unauthorized access, misuse and abuse of the system. This is achieved primarily by analyzing security-related events — which are recorded in the so-called *audit trails* — as they occur in the system.

In this paper, we propose an *integrated approach* by adding a configuration analysis component to our previous work on automated audit trail analysis and intrusion detection [5, 17]. By continuously monitoring user actions altering the protection configuration, the integrated system has the ability to constantly adapt itself to the current state of the configuration. It not only reports newly created holes in real time, but it also triggers appropriate detection rules to watch for attacks exploiting them.

The rest of this paper is organized as follows. Section 2 reviews related works on configuration analysis and intrusion detection. In Section 3, we outline the architecture of the integrated system, we recall the main features of our intrusion detection system ASAX, and we describe the novel deductive component devoted to configuration analysis. Section 4 depicts a detailed example illustrating the functionality of the system. In Section 5 we present some preliminary performance measurements that bear out the feasibility of the proposed approach. Lastly, Section 6 draws conclusions and outlines our future works.

2. Related Works

Configuration analysis systems perform a systematic analysis of computer configurations in order to uncover possible security breaches that can make the system vulnerable to break-ins, misuse or abuse. For instance, access control mechanisms may be useless if security critical files or directories contain vulnerabilities or are not adequately protected against malicious corruption. Several configuration analysis tools have been developed and are used periodically to assess the system vulnerability to attacks. For example, the computer oracle password and security system (COPS) [3] reports a list of common vulnerabilities such as weak passwords, world writable user home directories, world writable security-relevant files and directories, unexpected setUID files, improper exports of file systems, etc. A major component of COPS is the SU-KUANG [1] system which is a rule-based security checker system that adopts an attacker's reasoning in searching for security holes in the protection configuration. The administrator specifies an attacker goal (e.g., obtain root access to the system) and a set of initial privileges that are granted to the attacker. Using simple backward chaining, it recursively expands the attacker's goal into one or more subgoals. If the attacker's goal can be eventually reduced to a set of basic facts about the assumed privileges and the actual configuration, a potential attack is found. Moreover, the sequence of actions leading to the attacker's goal are reported. This simple rule-based system has proven very effective in finding security breaches in a Unix configuration and is periodically ran by many administrators to check their systems. NetKuang [24] is a multi-host configuration vulnerability checker that extends the functionalities of SU-KUANG [1] to a network environment. The security administrator tool for analyzing networks (SATAN) [22] and the internet scanner [9] have been developed to aid in identifying vulnerable network services in a network domain. Other similar systems have also been developed such as Miro [8] and Tripwire [11].

In the area of *intrusion detection*, a great deal of research effort has been devoted to establishing the foundations of intrusion detection technology [2, 13] and a few operational systems have been built [5, 7, 12, 14, 23]. IDES systems incorporate two main detection techniques. *Anomaly detection* techniques consist of maintaining a normal statistical profile for each user and comparing such a profile with the current user's behavior. User profiles store predefined measures describing aspects of users' behavior such as program execution, file access, CPU usage, etc. Any significant deviation with respect to the historical profile is considered abnormal and should raise suspicion. On the other hand *rule-based detection* techniques are concerned with detecting known suspicious behaviors and attack patterns as encoded in detection rules. Attack scenarios are detected by pattern-matching audit records in the audit trail against the a priori attack scenario. In case an anomaly or a known attack scenario is detected, the IDES system takes *ad-hoc* measures such as notifying the security administrator of the incident. In some cases, an IDES system can respond actively to intrusions by stopping offending processes, closing sessions, disabling accounts, etc. Neural networks are also used as an approach to intrusion detection (see [10]).

3. Integrating Configuration Analysis and Intrusion Detection

In this section, we propose an integrated system that extends our intrusion detection system ASAX [4, 5, 6]. ASAX (*Advanced Security audit trail Analysis on uniX*) features the rule-based language RUSSEL specifically designed for efficient, real time, analysis of audit trails. Recently, we were inspired by Baldwin's SU-KUANG system and we developed a hand-made, datalog-like language allowing us to represent the reasoning of an attacker in searching for security holes, in much the same way as SU-KUANG. However, contrary to SU-KUANG, our system is permanently active and it thus provides a continuous assessment of the software configuration.

The rest of this section is organized as follows: Section 3.1 is an overview of the integrated system. Section 3.2 reviews the main features of ASAX and provides an example detection rule in the RUSSEL language. (A comprehensive description of ASAX can be found in [4, 5, 6].) In Section 3.3 we describe the novel configuration analysis subsystem.

3.1. Overview of the Integrated System (see Figure 1)

Upon receiving an event from the audit subsystem, the intrusion detection subsystem executes all currently active detection rules. If the current audit record represents a malicious action, these rules may send alarm messages to alert the security administrator. If the current audit record reports a modification of some security sensitive file, one of the current rules executes a routine that accordingly updates the fact base describing the access control configuration. Additionally, the current rules may trigger new rules for the next record. Afterwards, the configuration analysis subsystem *incrementally* updates the current fact base to accommodate all logical consequences of the last analyzed event. This is done in two steps. First, since the current event may imply that some access rights to security sensitive files are

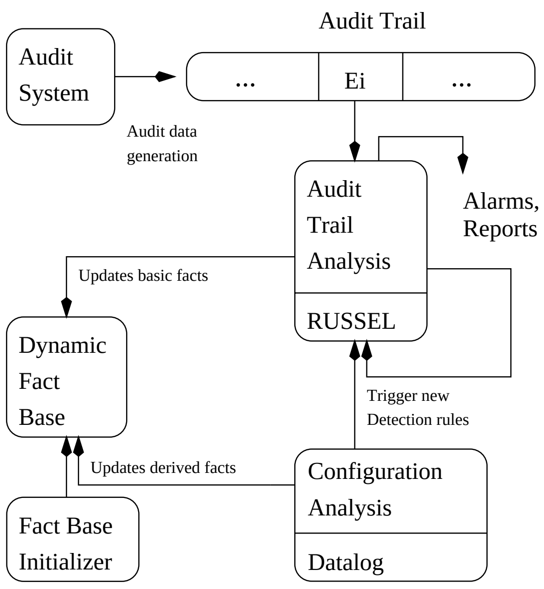

Figure 1. Combining Audit Trail Analysis and Configuration Analysis

retracted, the corresponding facts are retracted from the fact base as well as all derived facts implied by the deductive rules. Second — and similarly —, as new access rights may be granted for some files, all corresponding basic and derived facts are added to the fact base. Finally, as a consequence of the modifications to the fact base, the intrusion detection component may deactivate irrelevant detection rules and may trigger off some other ones in order to watch for possible users exploiting any newly created breach.

Initialization of the integrated system is performed by the fact base initializer, which builds the fact base reflecting the start-up configuration.

3.2. The Intrusion Detection System ASAX

3.2.1 Normalized Audit Trails

The intrusion detection system ASAX is theoretically able to analyze arbitrary sequential files and, more generally, any kind of data streams such as, for instance, network packets. This is achieved by translating the native file into a canonical format called NADF (*Normalized Audit Data Format*)[16]. According to this format, each audit data field in the audit record is converted to three fields: *identifier, length* and *value*. A leading 4-bytes integer represents the length of the whole NADF record. Note that the NADF format is similar to the TLV (*Tag, Length, Value*) encoding used for the BER (*Basic Encoding Rules*) which is used as part of the *Abstract Syntax Notation* ASN.1 [20]. As a result, the flexibility of the NADF format allows us to perform a straightforward translation of native files and a fast processing of NADF records through the RUSSEL language. Note also that every normalized audit data is given a symbolic name in order to be processed by the RUSSEL language.

3.2.2 The RUSSEL Language

RUSSEL (*RUle-baSed Sequence Evaluation Language*) is a novel language specifically tailored to the problem of searching arbitrary patterns of records in sequential files. The built-in mechanism of rule triggering allows a single pass analysis of the sequential file *from left to right*.

The language provides common control structures such as conditional, repetitive, and compound actions. Primitive actions include assignment, external routine call, and rule triggering. A RUSSEL program simply consists of a set of rule declarations which are made of a rule name, a list of formal parameters and local variables, and an action part. RUSSEL also supports modules sharing global variables and exported rule declarations.
The operational semantics of RUSSEL can be sketched as follows:

- records are analyzed sequentially. The analysis of the current record consists of executing all active rules. The execution of an active rule may alter global variables, trigger off new rules, raise alarms, send report messages to the security administrator, etc;

- rule triggering is a special mechanism by which a rule is made active either for the current or the next record. Typically, a rule is active for the current record because a prefix of a particular sequence of audit records has been detected. (The rest of this sequence has still to be possibly found in the rest of the file.) Actual parameters in the set of active rules represent knowledge about the already found subsequence and is useful for selecting further records in the sequence;

- when all the rules that are active for the current record have been executed, the next record is read and the rules triggered for it in the previous step are executed in turn;

- to initialize the process, a set of so-called *init rules* are made active for the first record.

User-defined and built-in C-routines can be called from a rule body. A simple and clearly specified interface with C allows the user to extend the RUSSEL language with any desirable feature. This includes simulation of complex data structures, sending an alarm message to the security officer, locking an account in case of outright security violation, etc.

3.2.3 Efficiency of ASAX

Efficiency is a critical requirement for the analysis of large sequential files, especially when on-line monitoring is involved. RUSSEL is efficient since it applies a bottom-up approach in constructing the searched record patterns. Furthermore, optimization issues are carefully addressed in the implementation of RUSSEL: for instance, the internal code generated by the compiler ensures a fast evaluation of boolean expressions and the current record is pre-processed before evaluation by all the current rules, in order to provide a direct access to its fields.

3.2.4 Distributed Audit Trail Analysis

A distributed version of the automated audit trail analyzer has been developed that allows us to perform a real-time analysis of multiple distributed audit trails originating from different machines, gathered at a central host [17].

3.2.5 An Example: Detecting Access to Critical Files

The use of the RUSSEL language for audit trail analysis is best illustrated by the example depicted in Figure 2. This example aims at detecting user actions altering the state of the protection configuration. The rule `criticalFileAccess` examines each record of the audit trail in sequence. If the current record involves an access to a security critical file or directory or to any of its parent directories (e.g., `/etc/passwd`, `/var/spool`, or a user start-up files such as `.cshrc`), the external routine `update_fact_base` invokes the configuration component which updates the current fact base. For instance, if the current audit record reports the creation of the start-up file `~amo/.cshrc` which grants write access to all users, the facts `owner(amo, ~amo/.cshrc)` and `worldWrite(~amo/.cshrc)` are added (among other facts) to the fact base. If a subsequent record indicates that the permission modes of the file `~amo/.cshrc` have been changed so that write access is only granted to the members of the group `folon`, the fact `worldWrite(~amo/.cshrc)` is retracted from the fact base while the fact `groupWrite(~amo/.cshrc, folon)` is added. Moreover, when a fact is added to the fact base, the deductive subsystem also adds all facts that are logical consequences of the added fact. Similarly, when a fact is retracted form the fact base, the subsystem retracts all its logical consequences (see 3.3). After analyzing the current record, the rule retriggers itself for the next record so that the rule will apply to every subsequent record in the audit trail. At the beginning of the analysis, the special rule **init_action** triggers the rule `criticalFileAccess` for the first record, and also executes the routine `init_fact_base`. The latter

```
rule criticalFileAccess;
begin
if
  ret_err = 0               /* success   */
  and (event = 4            /* creat(2)  */
    or event = 6            /* unlink(2) */
    or event = 10           /* chmod(2)  */
    or event = 39           /* fchmod(2) */
    or event = 42           /* rename(2) */
    or (73 <= event <= 83)) /* open(2)   */
  and (isPrefix(fname, '/etc/aliases')
    or isPrefix(fname, '/etc/group')
    or isPrefix(fname, '/etc/passwd')
    or isPrefix(fname,
             '/var/spool/cron/crontabs')
    or isStartUp(fname))
  -->
  updt_fact_base(event, fname, uid, gid, mode)
fi;
trigger off for_next criticalFileAccess
end;

init_action;
begin
    trigger off for_next criticalFileAccess;
    init_fact_base('datalog.log')
end.
```

Figure 2. Detecting Access to Critical Files

compiles the datalog program `datalog.log` and invokes the fact base initializer component, which builds the initial fact base by browsing the actual configuration.

3.3. The Configuration Analysis Subsystem

3.3.1 Deductive Language for Configuration Analysis

The goal of the deductive language for configuration analysis is to provide a simple formalism for representing the reasoning of an attacker trying to find holes in the configuration. The benefit of the deductive language is an increased flexibility in building and maintaining the attacker reasoning using predicate (Horn clause) logic. The integration of a deductive component in an IDES system is motivated by two facts. First, existing configuration analysis systems such as SU-KUANG do not prevent a malicious user from creating and closing holes in the system security between successive snapshots. Thus, these holes may go unnoticed by the system administrator if s/he relies solely on this type of systems. Second, an IDES system may look for attack scenarios that are irrelevant because the configuration is in fact not vulnerable to these attacks. Our language is based on a restricted (datalog like) form of predicate logic and allows us to express how an attacker is able to extend his privileges by exploiting weaknesses in the protection configuration. Attacker goals are represented by atoms (e.g., stealing the identity of user `joe` by an attacker `attacker` is simply represented by the atom `become(attacker, joe)`). Therefore, an atom has the form $p(arg_1, ..., arg_n)$ where p is a predicate symbol and $arg_1, ..., arg_n$ are (restricted) terms. A term is either a variable name (starting with an upper case letter) or a constant. For our specific purpose, constants are interpreted as being either a user name, a group name or a path name. A rule is of the form h :- $a_1, ..., a_n$ where $h, a_1, ..., a_n$ denote atoms. There are two kinds of predicates. Derived predicates correspond to an attacker goal such as `become(U, root)` or `write(User, /etc/passwd)`. Basic predicates describe the state of the configuration in terms of access permission modes of files, directory structure, file ownership, etc (e.g., `worldWritable(/home/users/amo)`). A program in our language is a set of rules (see Figure 3). We also provide a clean interface between the deductive component and the RUSSEL language whereby the deductive component is able to modify the set of currently active detection rules as soon as a security hole is introduced in the protection configuration. In other words, the interface provides a mechanism allowing the integrated system to adapt the current detection rules to the security level as represented by the derived facts. Conversely, we added to the RUSSEL language a simple construct allowing detection rules to access the current fact base. As a result, a detection rule can deactivate itself if the fact base indicates that the current state of the configuration precludes the successful completion of the attack scenario it is attempting to detect. These two interfaces are supported as follows. On the deductive language side, we use the construct:

on_new *fact_name*($X_1, ..., X_n$)
trigger off for_next *rule_name*($X_1, ..., X_n$)

to trigger off for the next audit record the detection rule *rule_name*. On the RUSSEL language side, a rule uses the construct *is_fact*(*fact_name*(arg_1, ..., arg_n)) to lookup the fact base for the specified fact.

3.3.2 Implementation of the Inference Engine

The implementation of the inference engine has to be carefully addressed in order to meet the efficiency requirement. The main source of performance degradation is the process of updating the fact base upon reception of an audit record. The occurrence of an event drives the configuration from a previous state to a new state. The inference engine ensures that at each time, the content of the fact base represents the actual state of the configuration. In the following, we only provide an outline of the implementation, the reader is referred to [18] for a more detailed presentation of the implementation and of the logic programming aspects. There are two main operations that are carried out by the inference engine: fact base initialization and fact base incremental update. These operations are explained in the following.

3.3.3 Initializing the Fact Base

This operation consists of building the fact base by making a snapshot of the security critical pieces of the file system. This is similar to what is done in COPS [3] except that we represent security information in memory as a collection of basic facts. The next step is to compute all possible attacker goals from the datalog program. We make use of the *bottom-up semi-naive evaluation* algorithm (see [15, 19]) ensuring that all inferences are made only once. The algorithm terminates when no new attacker goals can be inferred from the rules and the fact base. At that time, the basic facts describe the state of the configuration while the derived facts (or attacker goals) describe the current privileges an attacker may acquire by exploiting the corresponding breaches in the configuration. Table 1 depicts a list of basic facts currently used and their meaning. In this table *F, D, U* and *G* refers to a file, a directory, a user and a group, respectively.

Basic Fact	*Meaning*
worldWrite(F)	F is writable by any user
groupWrite(F, G)	F is writable by the members of G
groupMember(U, G)	U is a member of G
parentDir(D, F)	D is the parent directory of F
start_up(U, F)	F is a start-up file of U
owner(U, F)	F is a owned by U

Table 1. Basic Facts

It should be stressed that only security-critical files are considered in collecting basic facts. For instance, the file `/etc/passwd` is described (among other basic facts) by the basic facts `parentDir(/etc, /etc/passwd)` and `parentDir(/, /etc)`.

3.3.4 Updating the Fact Base

As previously mentioned, the fact base is updated incrementally each time a relevant audit record is processed. Let *D* (resp. *A*) be the set of basic facts that must be discarded (resp. added) from the fact base due to the occurrence of the current event. The incremental update is performed in two steps as follows.

Removing Obsolete Facts In this step, we first remove from the fact base all basic facts in *D*. In addition we remove all attacker goals that were derived from those basic facts. Recursively, if other attacker goals were derived from removed goals, they are discarded as well. We repeat this process until we remove all attacker goals that are no longer implied by any deduction. (Note that an attacker goal can be implied by several different deductions.)

Adding New Facts During this step, we use an incremental fixpoint algorithm (see [15, 19]) to compute all newly derived attacker goals from the fact base obtained in the previous step and the set of new basic facts *A*. Additionally, we store the corresponding deductions in order to keep track of which attacker goals were derived from which other facts.

3.3.5 Low Level Implementation

In order to make the incremental algorithms efficient, we need to store every fact and every deduction into separate data structures. Each fact provides a direct access to all deductions to which it contributes while every deduction provides an access to the fact resulting from the deduction. In addition, each predicate provides access to all rules which it appears in the body of. Moreover, every fact contains a counter of the number of deductions from which the fact results. Unicity of representation of every fact and deduction is ensured by means of hashing techniques.

4. A Detailed Example

In this example we show how the deductive language is used to model the reasoning of an attacker searching for a security breach in the protection configuration of a Unix system. Additionally, we illustrate how the detection rules are activated when a security breach is created and we provide an example of such a detection rule.

4.1. Attacker Program for Unix

Figure 3 depicts an example datalog program which models the main elements of the Unix protection configuration system. Procedure *become/2* says that a user is able to acquire the privileges of the super-user if s/he is able to replace the file `/etc/passwd, /etc/aliases` or `/var/spool/cron/crontabs`. It also says that if a user is able to replace the start-up file of another user, then s/he is able to acquire the privileges of (or ”become”) that user. Procedure *replace/2* means that a user is able to replace a file if s/he is able to write to this file or if s/he is able to replace its parent directory. Procedure *write/2* says that a file can be written by a user if the file is world writable or if s/he is the owner of that file or if s/he can become a member of the group granting write access to that file. Finally, procedure *inGroup/2* says that a user is able to become a member of a group if s/he is a legitimate member of that group or if s/he can become a legitimate member of that group or if s/he can replace the file `/etc/group`. The last line in the program

```
become(U, root) :-
    replace(U, /etc/passwd).
become(U, root) :-
    replace(U, /etc/aliases).
become(U, root) :-
    replace(U, /var/spool/cron/crontabs).
become(U, V) :-
    start_up(V, F), replace(U, F).
replace(U, F) :-
    write(U, F).
replace(U, F) :-
    parentDir(D, F), replace(U, D).
write(U, F) :-
    owner(V, F), become(U, V).
write(U, F) :-
    groupWrite(F, G), inGroup(U, G).
write(U, F) :-
    worldWrite(F).
inGroup(U, G) :-
    groupMember(U, G).
inGroup(U, G) :-
    groupMember(V, G), become(U, V).
inGroup(U, G) :-
    replace(U, /etc/group).

on_new become(U, root) trigger off for_next
detect_root_access(U).
```

Figure 3. Rules for modeling Unix

means that the detection rule `detect_root_access` is triggered whenever an arbitrary user is potentially able to have root access. Each time the inference engine updates the fact base, it checks for the presence of the derived fact of the form `become(U, root)` where *U* is any user name. If such a fact is newly derived, the deductive component triggers the detection rule `detect_root_access` for the next audit record with the involved user name as an argument.

The detection rule `detect_root_access` is shown in Figure 4. It assumes that the attacker tries to obtain root access to the system through the execution of a setUID program (e.g., a copy of the shell `/bin/sh` under the directory `/tmp`) used as a back-door. The detection of this attack is performed as follows. If the current audit record corresponds to the execution of a suspicious setUID root program by the attacker, the rule reports the incident, in real-time. The rule remains active as long as the fact base contains the attacker goal `become(Username, root)`. Otherwise, the rule deactivates itself. (This is the case for instance when one of the basic facts which contributed to the derivation of the attacker goal `become(Username, root)` has been

```
rule detect_root_access(Username:  string);
begin
if
  (event = 7 or event = 23)
  /* exec(2) or execve(2) */
  and file_owner_id = 0 /* root */
  and uid = uid(Username)
  and isIllegalSetUID(fname)
  -->
  println('Suspicious Execution of the
           setUID program', fname,
           ' By User ', Username,
           ' At Time ', gettime(time))
fi;
if
  is_fact(become(Username, 'root'))
  -->
  trigger off for_next detect_root_access(Username)
fi
end.
```

Figure 4. Detecting Root Access

discarded from the fact base.) Furthermore, it is possible to report the chain of privileges gained by the attacker. This can be implemented straightforwardly since we keep track of which basic facts have contributed to a given attacker goal.

Note that there are as many active detection rules `detect_root_access` as there are attacker goals `become(Username, root)` with a different user name. Each of these active rules searches the audit trail for an occurrence of the above attack scenario performed by the given user name argument. The number of these active rules may be high if there are numerous users corresponding to the attacker goal `become(Username, root)`. However, one may detect all users performing the above attack scenario using a single active rule. This obtained by adding the inference rule `rootAccess :- become(U, root)` to the program in Figure 3 where the lines

```
on_new become(U, root)
trigger off for_next detect_root_access(U).
```

are replaced by

```
on_new rootAccess
trigger off for_next detect_root_access.
```

In this version, since `detect_root_access` has no argument, the condition *uid* = `uid(Username)` is no longer needed. However, in this version, `isIllegalSetUID(`*fname*`)` is evaluated much more frequently that in the first version where it is evaluated only if condition *uid* = `uid(Username)` fails. It is difficult to decide which of the two versions is more efficient since in the first version we have several active rules which are likely to be efficient, whereas in the second version, we have a single rule that is expected to be less efficient because the evaluation of `isIllegalSetUID(`*fname*`)` amounts to check if *fname* is a member of a long list of legal setUID programs in the file system.

Accuracy of the Fact Base

To ensure the reliability of the continuous monitoring, the fact base must correctly describe the actual state of the configuration. To ensure this, we must preselect for auditing all events susceptible to alter the state of the configuration. A challenging difficulty arises in an NFS environment wherein a user in a remote machine can mount a file system from the monitored host (which is then the NFS server). In this case, some inconsistencies may be introduced if the user modifies a security critical file or directory in the exported file system. This problem is addressed by deploying the distributed version of ASAX [17] on all NFS client hosts while monitoring the server. An audit record involving the modification of a security sensitive file in the mounted file system is sent to the central host where the fact base is updated accordingly. Alternatively, if the distributed auditing of all hosts is not possible, we can reinitialize the fact base periodically. Note that this also can be done incrementally.

A Note About the Languages

The reader may wonder why the languages respectively used for intrusion detection and for configuration analysis are so radically different. The reason is that they perform entirely different tasks: the RUSSEL language has to detect (possibly many) sequence patterns in an audit trail, in parallel, while the configuration analysis language is used for maintaining a set of logical consequences from an (evolving) set of basic facts. Thus, since efficiency is critical for a real time analysis, we have chosen to design two (communicating) languages especially crafted to the tasks. Other approaches are of course possible — for instance, using a logic language for both tasks — but we prefer to follow a more pragmatic approach, which we believe easier to implement efficiently.

5. Performance Measurements

This section reports preliminary performance tests of our integrated system. The first objective of these tests is to support our belief that our approach of real-time continuous assessment of the protection configuration is feasible in a real life context. The second objective is to show that the integration of the deductive component with the intrusion detection system ASAX can reduce the total computational effort for analyzing audit trails.

The experiment was carried out on a SPARCstation-5 with 32 Megabytes of RAM running the Solaris 2.5 operating system with the Basic Security Module [21] enabled. We collected audit data for a total time period of 48 hours, 42 minutes and 51 seconds. For a period of 6 hours, 51 minutes and 48 seconds within this total time period, we used a pseudo-random generator of security-related events in order to simulate global users' activity. Every three seconds, the event generator program (executed as *root*) randomly selects either a global security-sensitive object (such as `/etc/passwd` or `/var/spool/cron`), or a user's start-up file. Then the program randomly changes permission modes of the selected file such that write access is allowed or not allowed for the members of the group, for all users, or both.

Table 2 summarizes the characteristics of the audit data that is used in the analysis. It shows the number of users in the configuration (#U), the number of groups (#G), the total number of analyzed audit records (#Rec), the number of audit records involving a security-related object (#SRec), the size in Megabytes of the generated audit trail (Size), the time period covered by the audit trail (Time) and the average number of audit records generated per second (Rate).

#U	*#G*	*#Rec*	*#SRec*	*Size*	*Time*	*Rate*
				Mb	hh:mm:ss	
125	23	205,444	8236	32	48:42:51	1.18

Table 2. Audit Trail Description

We ran our test on the example presented in section 4 using the program in Figure 3 and the RUSSEL detection rule in Figure 4. Table 3 depicts various measures related to the fact base. It shows the total number of facts (#Facts) and deductions (#Deds) after initializing the fact base, the total amount of memory space (Size) occupied by these facts and deductions (in Kbytes), the CPU time (IFB) for initializing the fact base from scratch (in seconds), the total CPU time (UFB) for updating the fact base (in seconds), and the average CPU time (UPR) for updating the fact base per audit record (in milliseconds). These results suggest that continuous assessment of a software configuration with our system is feasible since the time devoted to incremental deduction is negligible *wrt* the time separating the generation of two successive audit records.

Lastly, we conducted a test comparing the performance benefit of combining our ASAX intrusion detection system with the deductive component. We measured the user, system and total CPU time required by the detection rule `detect_root_access` in Figure 4 for the analysis of the audit trail described in Table 2. In one case (first line in Table 4), the rule uses the deductive component while in the second case, it does not. These results show a sensible performance gain thanks to the fact that the detection rule `detect_root_access` is triggered only when the attacker goal `become(U, root)` is added to the fact base. As soon as this goal is retracted from the fact base (which means that the root account is no more vulnerable), the detection rule `detect_root_access` deactivates itself. The attacker goal `become(U, root)` was added to the fact base while analyzing the 115383th audit record and was removed at the 125632th record. In other words, the rule was active for only 10,049 records over 205,444 total number of records.

#Facts	*#Deds*	*Size*	*IFB*	*UFB*	*UPR*
		Kb	*sec*	*sec*	*msec*
6929	11037	862	80.4	138.4	0.67

Table 3. Fact Base Description

type	*usr*	*sys*	*total*	*#RPS*
	sec	*sec*	*sec*	
Integrated	339	97	436	471
Not Integrated	349	97	446	460

Table 4. Integrated v.s. Non Integrated Analysis

The performance benefit is not significant since the intrusion detection component uses only one detection rule. However, this performance gain is expected to be higher if we use several detection rules at the same time. In this case, the reduction of the CPU cost for detection rules will outweigh the overhead of maintaining the fact base.

Finally, from Table 4, we can see that (#RPS) the number of audit records processed per second is approximately 400 times larger that the number of records generated per second (1.18), which once again supports the feasibility of the real time assessment.

6. Conclusions and Future Works

In this paper, we have presented the deductive component of a programming system that aims at analyzing a computer configuration as well as its users' behaviors efficiently (in real time). We use a specific —hand-made— implementation of the logic language since we believe that such a specialized implementation is needed to attain our efficiency goals. Future works will be devoted to assess the practicality of our system for monitoring various aspects of network management and computer security in a real life context. In particular, we will develop methods to program and tune such a system systematically.

Effectiveness of the integrated system was supported by reporting performance measurements conducted on a simulated example. These measurements seem to suggest that on-line continuous analysis is feasible.

References

[1] Robert W. Baldwin. Kuang: Rule-based Security Checking. Technical report, MIT, Lab for Computer Science Programming Systems Research Group, May 1994.

[2] D. Denning. An Intrusion-Detection Model. *IEEE Trans. Softw. Eng.*, 13(2), Feb. 1987.

[3] D. Farmer and E.H. Spafford. COPS Security Checker System. In *Proceedings of the Summer Usenix Conference*, Pages 165-170, 1990, Anaheim CA. ftp://coast.cs.purdue.edu/pub/Purdue/papers/spafford/farmer-spaf-cops.ps.Z

[4] N. Habra, B. Le Charlier and A. Mounji. Preliminary Report on Advanced Security Audit Trail Analysis on Unix. Technical report, University of Namur, Institut d'Informatique, Rue Grandgagnage, 21 B-5000 Namur, Belgium, Dec. 1991. http://www.info.fundp.ac.be/˜amo/papers/spec.ps.Z.

[5] N. Habra, B. Le Charlier, A. Mounji and I. Mathieu. ASAX: Software Architecture and Rule-based Language for Universal Audit Trail Analysis. In *Proceedings of the Second European Symposium on Research in Computer Security (ESORICS). Toulouse, France*. Springer-Verlag, Nov. 1992. http://www.info.fundp.ac.be/˜amo/papers/esorics92.ps.Z.

[6] N. Habra, B. Le Charlier and A. Mounji. Advanced Security Audit Trail Analysis on Unix. Implementation Design of the NADF Evaluator. Technical report, University of Namur, Institut d'Informatique, Rue Grandgagnage, 21 B-5000 Namur, Belgium, Mar. 1993. http://www.info.fundp.ac.be/˜amo/papers/design.ps.Z.

[7] P. Helman, G.E. Liepins and B. Rochards. Foundations of Intrusion Detection. In *Proceedings of the Fifth Computer Security Foundations Workshop*, June 1992.

[8] A. Heydon. Specifying and Checking Unix Security Constraints. In *Proceedings of the 3rd USENIX Security Symposium*, Sept. 1992.

[9] Internet Scanner. Technical report, Internet Security Systems, 1995. http://www.iss.net/iss/scanner.html

[10] L. Kevin. A Neural Network Approach Towards Intrusion Detection. In *Proceedings of the 13th National Computer Security Conference*, pages 125–134, Washington, DC, Oct. 1990.

[11] G. Kim and E. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. In *Proceedings of the Systems Administration, Networking and Security Conference III (SANS)*, Washington DC, 1994. ftp://coast.cs.purdue.edu/pub/COAST/papers/Tripwire.ps.Z.

[12] Sandeep Kumar. Classification and Detection of Computer Intrusions. *PhD Thesis*, Department of Computer Science, Purdue University, West Lafayette, IN 47907, Nov. 1995. ftp://coast.cs.purdue.edu/pub/COAST/papers/kumar-intdet-phddiss.ps.Z.

[13] T. F. Lunt. Automated Audit Trail Analysis and Intrusion Detection: A Survey. In *Proceedings of the 11th National Security Conference*, Baltimore, MD, Oct. 1988.

[14] T. F. Lunt and R. Jagannathan. A Prototype Real-Time Intrusion Detection Expert System. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, Apr. 1988.

[15] M. J. Maher and R. Ramakrishnam. Déjà vu in Fixpoints of Logic Programs. Technical report, Computer Sciences Department, University of Wisconsin-Madison, Nov. 1989.

[16] A. Mounji. User Guide for Implementing NADF Adaptors. Technical report, University of Namur, Institut d'Informatique, Rue Grandgagnage, 21 B-5000 Namur, Belgium, Jan. 1995. http://www.info.fundp.ac.be/˜amo/papers/nadf.ps.Z.

[17] A. Mounji, B. Le Charlier, D. Zampunieris and N. Habra. Distributed Audit Trail Analysis. In *Proceedings of the ISOC'95 Symposium on Network and Distributed Systems Security*. IEEE Computer Society Press, 10662 Los Vaqueros Circle/P.O. Box 3014, Los Alamitos, CA 90720-1264, Feb. 1995. http://www.info.fundp.ac.be/˜amo/papers/isoc95.ps.Z.

[18] A. Mounji and B. Le Charlier. Detecting Breaches in Computer Security: A Pragmatic System with a Logic Programming Flavor. In *Proceedings of the 8th Benelux Workshop on Logic Programming*, Louvain-La-Neuve, Belgium, Sept. 1996. http://www.info.fundp.ac.be/˜amo/papers/benelog96.ps.Z.

[19] Raghu Ramakrishnam, Divesh Srivastava and S. Sudarshan. Efficient Bottom-up Evaluation of Logic Programs. Technical report, Computer Sciences Department, University of Wisconsin-Madison, Nov. 1989.

[20] M. T. Rose. *The Open Book: a Practical Perspective on OSI.* Prentice-Hall. ISBN 0-13-643016-3, Sept. 1990.

[21] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. *SunSHIELD Basic Security Module Guide*, Aug. 1994.

[22] W. Venema. Project Satan: Unix/Internet Security. In *Proceedings of the COMPSEC'95 conference*, Elsevier, London, 1995. http://www.fish.com/zen/satan/satan.html.

[23] J. Winkler. Unix Prototype for Intrusion and Anomaly Detection in Secure Networks *Planning Research Corporation, R&D.* 1990.

[24] D. Zerkle and K. Levitt. NetKuang — A Multi-Host Configuration Vulnerability Checker. In *6th USENIX Security Symposium*, San Jose, California, July 1996. http://seclab.cs.ucdavis.edu/papers/zl96.ps.